CS-XXX: Graduate Programming Languages

Lecture 20 — Shared-Memory Parallelism and Concurrency

Dan Grossman
2012

Concurrency and Parallelism

- PL support for concurrency/parallelism a huge topic
  - Increasingly important (not traditionally in PL courses)
  - Lots of active research as well as decades-old work
- We'll just do *explicit threads* plus:
  - Shared memory (*locks* and *transactions*)
  - Futures
  - Synchronous message passing (*Concurrent ML*)
- We'll skip
  - Process calculi (foundational message-passing)
  - Asynchronous methods, join calculus, ...
  - Data-parallel languages (e.g., NESL or ZPL)
  - ...
- Mostly in ML syntax (inference rules where convenient)
  - Even though current OCaml implementation has threads but not parallelism

Concurrency vs. Parallelism

(Terminology not universal, but distinction paramount):

Concurrency *is about correctly and efficiently managing access to shared resources*

- Examples: operating system, shared hashtable, version control
- Key challenge is responsiveness to external events that may arrive asynchronously and/or simultaneously
- Often provide responsiveness via threads
- Often focus on *synchronization*

Parallelism *is about using extra computational resources to do more useful work per unit time*

- Examples: scientific computing, most graphics, a lot of servers
- Key challenge is Amdahl's Law (no sequential bottlenecks)
- Often provide parallelism via threads on different processors and/or to mask I/O latency

Threads

High-level: "Communicating sequential processes"

Low-level: "Multiple stacks plus communication"

From OCaml's `thread.mli`:

```
type t (*thread handle; remember we're in module Thread*)
val create : ('a->'b) -> 'a -> t (* run new thread *)
val self : unit -> t (* what thread is executing this? *)
```

The *code* for a thread is in a closure (with hidden fields) and `Thread.create` actually *spawns* the thread

Most languages make the same distinction, e.g., Java:

- Create a `Thread` object (data in fields) with a `run` method
- Call its `start` method to actually spawn the thread

Why use threads?

One *OR* more of:

1. Performance (multiprocessor *or* mask I/O latency)
2. Isolation (separate errors *or* responsiveness)
3. Natural code structure (1 stack awkward)

It's not just performance

On the other hand, it seems fundamentally harder (for programmers, language implementors, language designers, semanticists) to have multiple threads of execution

One possible formalism (omitting thread-ids)

- Program state is one heap and multiple expressions
- Any e_i might "take the next step" and potentially spawn a thread
- A value in the "thread-pool" is removable
- Nondeterministic with *interleaving granularity* determined by rules

Some example rules for $H; e \to H'; e'; o$ (where $o ::= \cdot \mid e$):

$$\frac{}{H; !l \to H; H(l); \cdot} \qquad \frac{H; e_1 \to H'; e_1'; o}{H; e_1\ e_2 \to H'; e_1'\ e_2; o}$$

$$\frac{}{H; \mathsf{spawn}(v_1, v_2) \to H; 0; (v_1\ v_2)}$$

Formalism continued

The $H; e \to H'; e'; o$ judgment is just a helper-judgment for $H; T \to H'; T'$ where $T ::= \cdot \mid e; T$

$$\frac{H; e \to H'; e'; \cdot}{H; e_1; \ldots; e; \ldots; e_n \to H'; e_1; \ldots; e'; \ldots; e_n}$$

$$\frac{H; e \to H'; e'; e''}{H'; e_1; \ldots; e; \ldots; e_n \to H'; e_1; \ldots; e'; \ldots; e_n; e''}$$

$$\frac{}{H; e_1; \ldots; e_{i-1}; v; e_{i+1}; \ldots; e_n \to H; e_1; \ldots; e_{i-1}; e_{i+1}; \ldots; e_n}$$

Program termination: $H; \cdot$

Equivalence just changed

Expressions equivalent in a single-threaded world are not necessarily equivalent in a multithreaded context!

Example in OCaml:

```
let x, y = ref 0, ref 0 in
create (fun () -> if (!y)=1 then x:=(!x)+1) ();
create (fun () -> if (!x)=1 then y:=(!y)+1) () (* 1 *)
```

Can we replace line (1) with:

```
create (fun () -> y:=(!y)+1; if (!x)<>1 then y:=(!y)-1) ()
```

For more compiler gotchas, see “Threads cannot be implemented as a library” by Hans-J. Boehm in PLDI2005

- Example: C bit-fields or other adjacent fields

Communication

If threads do nothing other threads need to “see,” we are done

- Best to do as little communication as possible
- E.g., do not mutate shared data unnecessarily, or hide mutation behind easier-to-use interfaces

One way to communicate: Shared memory

- One thread writes to a ref, another reads it
- Sounds nasty with pre-emptive scheduling
- Hence synchronization mechanisms
  - Taught in O/S for historical reasons!
  - Fundamentally about restricting interleavings

Join

“Fork-join” parallelism a simple approach good for “farm out subcomputations then merge results”

```
(* suspend caller until/unless arg terminates *)
val join : t -> unit
```

Common pattern:

```
val fork_join : (’a -> ’b array) -> (* divider *)
                (’b -> ’c) ->       (* conqueror *)
                (’c array -> ’d) -> (* merger *)
                ’a ->               (* data *)
                ’d
```

Apply the second argument to each element of the `’b array` in parallel, then use third argument *after* they are done

See `lec20code.ml` for implementation and related patterns (untested)

Futures

A different model for explicit parallelism without explicit shared memory or message sends

- Easy to implement on top of either, but most models are easily inter-implementable
- See ML file for implementation over shared memory

```
type ’a promise;
val future : (unit -> ’a) -> ’a promise (*do in parallel*)
val force : ’a promise -> ’a (*may block*)
```

Essentially fork/join with a value returned?

- Returning a value more functional
- Less structured than “cobegin s1; s2; ... sn” form of fork/join

Locks (a.k.a. mutexes)

```
(* mutex.mli *)
type t (* a mutex *)
val create : unit -> t
val lock   : t -> unit (* may block *)
val unlock : t -> unit
```

OCaml locks do not have two common features:

- Reentrancy (changes semantics of `lock` and `unlock`)
- Banning nonholder release (changes semantics of `unlock`)

Also want condition variables (`condition.mli`), not discussed here

Using locks

Among infinite correct idioms using locks (and more incorrect ones), the most common:

- Determine what data must be "kept in sync"
- Always acquire a lock before accessing that data and release it afterwards
- Have a *partial order* on all locks and if a thread holds m_1 it can acquire m_2 only if $m_1 < m_2$

See canonical "bank account" example in `lec20code.ml`

Coarser locking (more data with same lock) trades off parallelism with synchronization

- Under-synchronizing the hallmark of concurrency incorrectness
- Over-synchronizing the hallmark of concurrency inefficiency

The Evolution Problem

Write a new function that needs to update $o1$ and $o2$ together.

- What locks should you acquire? In what order?
- There may be no answer that avoids *races* and *deadlocks* without breaking old code. (Need a stricter partial order.)
  - Race conditions: See definitions later in lecture
  - Deadlock: Cycle of threads blocked forever

See `xfer` code in `lec16code.ml`

Real Java example:

```
synchronized append(StringBuffer sb) {
 int len = sb.length(); //synchronized
 if(this.count+len > this.value.length) this.expand(...);
 sb.getChars(0,len,this.value,this.count); //synchronized
 ...
}
```

Undocumented in 1.4; in 1.5 caller synchronizes on `sb` if necessary

Atomic Blocks (Software Transactions)

Java-like: `atomic { s }`

OCaml-like: `atomic : (unit -> 'a) -> 'a`

Execute the body/thunk *as though* no interleaving from other threads

- Allow parallelism unless there are actual run-time memory conflicts (detect and abort/retry)
- Convenience of coarse-grained locking with parallelism of fine-grained locking (or better)
- But language implementation has to do more to detect conflicts (much like garbage collection is convenient but has costs)

Most research on implementation (preserve parallelism unless there are conflicts), but this is not an implementation course

Transactions make things easier

Problems like `append` and `xfer` become trivial

So does mixing coarse-grained and fine-grained operations (e.g., hashtable lookup and hashtable resize)

Transactions *are* great, but not a panacea:

- Application-level races can remain
- Application-level deadlock can remain
- Implementations generally try-and-abort, which is hard for "launch missiles" (e.g., I/O)
- Many software implementations provide a weaker and under-specified semantics if there are data races with non-transactions
- *Memory-consistency model* questions remain and may be worse than with locks...

Data races, informally

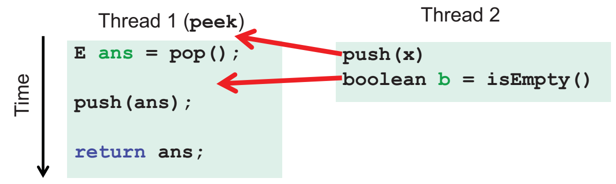
[More formal definition to follow]

"*race condition*" means two different things

- *Data race:* Two threads read/write, write/read, or write/write the same location without intervening synchronization
  - So two conflicting accesses could happen "at the same time"
  - Better name not used: *simultaneous access error*
- *Bad interleaving:* Application error due to thread scheduling
  - Different order would not produce error
  - A data-race free program can have bad interleavings

Bad interleaving example

```
class Stack<E> {
  … // state used by isEmpty, push, pop
  synchronized boolean isEmpty() { … }
  synchronized void push(E val) { … }
  synchronized E pop() { … }
  E peek() { // this is wrong
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

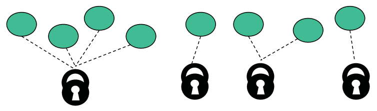


```
E ans = pop();

push(ans);

return ans;
```

```
push(x)
boolean b = isEmpty()
```

Consistent locking

If all mutable, thread-shared memory is *consistently guarded by some lock*, then data races are impossible



But:

- Bad interleavings can remain: programmer must make *critical sections* large enough
- Consistent locking is *sufficient* but not *necessary*
  - A tool detecting consistent-locking violations might report "problems" even if no data races are possible

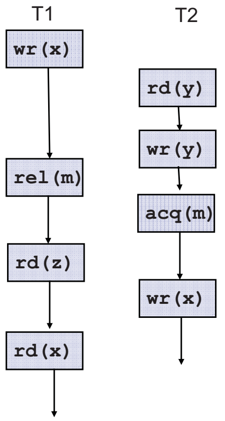
Data races, more formally

Let threads T1, …, Tn perform *actions*:

- *Read* shared location *x*
- *Write* shared location *x*
- [Successfully] *Acquire* lock *m*
- *Release* lock *m*
- Thread-local actions (local variables, control flow, arithmetic)
  - Will ignore these

Order in one thread is *program order*

- Legal orders given by language's single-threaded semantics + reads

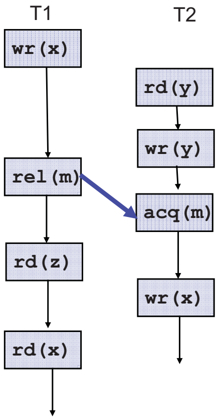


Data races, more formally

Execution [trace] is a partial order over actions *a1* < *a2*

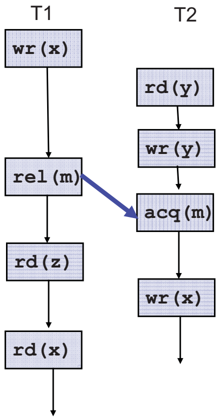
- *Program order:* If Ti performs *a1* before *a2*, then *a1* < *a2*
- *Sync order:* If a2=(Ti acquires m) occurs after a1=(Tj releases m), then *a1* < *a2*
- *Transitivity:* If *a1* < *a2* and *a2* < *a3*, then *a1* < *a3*

Called the *happens-before relation*



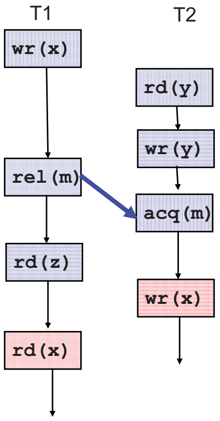
Data races, more formally

- Two actions *conflict* if they read/write, write/read, or write/write the same location
  - Different locations not a conflict
  - Read/read not a conflict



Data races, more formally

- Finally, a *data race* is two *conflicting* actions *a1* and *a2* *unordered* by the happens-before relation
  - *a1* ≮ *a2* and *a2* ≮ *a1*
  - By definition of happens-before, actions will be in different threads
  - By definition of conflicting, will be read/write, write/read, or write/write
- A program is *data-race free* if no trace on any input has a data race



Beyond locks

Notion of data race extends to synchronization other than locks

- Just define happens-before appropriately

Examples:

- Thread fork
- Thread join
- Volatile variables

Why care about data races?

Recall not all race conditions are data races…

So why focus on data races?

- One answer: Find some bugs without application-specific knowledge

- More interesting: Semantics for modern languages very *relaxed* for programs with data races
  - Else optimizing compilers and hardware too difficult in practice
  - Increases importance of writing data-race-free programs

An example

Can the assertion fail?

```
                // shared memory
                a = 0; b = 0;

// Thread 1        ||   // Thread 2
x = a + b;         ||   b = 1;
y = a;             ||   a = 1;
z = a + b;         ||
assert(z>=y);      ||
```

An example

Can the assertion fail?

```
                // shared memory
                a = 0; b = 0;

// Thread 1        ||   // Thread 2
x = a + b;         ||   b = 1;
y = a;             ||   a = 1;
z = a + b;         ||
assert(z>=y);      ||
```

- Argue assertion cannot fail:
  `a` never decreases and `b` is never negative, so `z>=y`
- But argument makes implicit assumptions you *cannot* make in Java, C#, C++, etc. (!)

Common-subexpression elimination

Compilers simplify/optimize code in many ways, e.g.:

```
                // shared memory
                a = 0; b = 0;

// Thread 1        ||   // Thread 2
x = a + b;         ||   b = 1;
y = a;             ||   a = 1;
z = ~~a + b;~~ x;     ||
assert(z>=y);      ||
```

Now assertion can fail

- As though third read of `a` precedes second read of `a`
- Many compiler optimizations have the effect of reordering/removing/adding memory operations like this (exceptions: constant-folding, function inlining, …)

A decision…

```
                // shared memory
                a = 0; b = 0;

// Thread 1        ||   // Thread 2
x = a + b;         ||   b = 1;
y = a;             ||   a = 1;
z = ~~a + b;~~ x;     ||
assert(z>=y);      ||
```

Language semantics ***must*** resolve this tension:

- If assertion can fail, the program is wrong
- If assertion cannot fail, the compiler is wrong

Memory-consistency model

- A *memory-consistency model* (or *memory model*) for a shared-memory language specifies *which write a read can see*
  - Essential part of language definition
  - Widely under-appreciated until last several years

- Natural, strong model is *sequential consistency (SC)* [Lamport]
  - Intuitive "interleaving semantics" with a global memory

> *"the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"*

Considered too strong

- Under SC, compiler is wrong in our example
  - Must disable any optimization that has effect of reordering memory operations [on mutable, thread-shared memory]
- So modern languages do *not* guarantee SC
  - Another reason: Disabling optimization insufficient because the hardware also reorders memory operations unless you use very expensive (10x-100x) instructions
- But still need *some* language semantics to reason about programs…

The "grand compromise"

- Basic idea:
  - Guarantee SC only for "*correctly synchronized"* programs [Adve]
  - Rely on programmer to synchronize correctly
  - Correctly synchronized == data-race free (DRF)!
- More precisely:

  If every SC execution of a program *P* has no data races,
  then every execution of *P* is equivalent to an SC execution
  - Notice we use SC to decide if *P* has data races
- Known as "DRF implies SC"

Roles under the compromise

- Programmer: write a DRF program
- Language implementor: provide SC assuming program is DRF

But what if there *is* a data race:

- C++: anything can happen
  - "catch-fire semantics"
  - Just like array-bounds errors, uninitialized data, etc.
- Java/C#: very complicated story
  - Preserve safety/security despite reorderings
  - "DRF implies SC" a theorem about the very-complicated definition

Back to the example

Code has a data race, so program is wrong and compiler is justified

```
            // shared memory
            a = 0; b = 0;

// Thread 1           ||   // Thread 2
x = a + b;            ||   b = 1;
y = a;                ||   a = 1;
z = ~~a + b;~~ x;        ||
assert(z>=y);         ||
```

Back to the example

This version is DRF, so the "optimization" is *illegal*

- Compiler would be wrong: assertion must not fail

```
                  // shared memory
                  a = 0; b = 0;
                  m a lock

// Thread 1            ||   // Thread 2
sync(m){x = a + b;}    ||   sync(m){b = 1;}
sync(m){y = a;}        ||   sync(m){a = 1;}
sync(m){z = a + b;}    ||
assert(z>=y);          ||
```

Back to the example

This version is DRF, but the optimization is *legal* because it does not affect *observable* behavior: the assertion will not fail

```
            // shared memory
            a = 0; b = 0;
            m a lock

// Thread 1           ||   // Thread 2
sync(m){              ||   sync(m){
 x = a + b;           ||     b = 1;
 y = a;               ||     a = 1;
 z = ~~a + b;~~ x;       ||   }
}
assert(z>=y);
```

Back to the example

This version is also DRF and the optimization is illegal

- Volatile fields (cf. C++ atomics) exist precisely for writing “clever” code like this (e.g., lock-free data structures)

```
// shared memory
volatile int a, b;
a = 0;
b = 0;
```

```
// Thread 1
x = a + b;
y = a;
z = a + b;
assert(z>=y);
```

```
// Thread 2
b = 1;
a = 1;
```

So what is allowed?

How can language implementors know if an optimization obeys “DRF implies SC”? Must be aware of threads! [Boehm]

Basically 2.5 rules suffice, *without* needing inter-thread analysis:

0. Optimization must be legal for single-threaded programs
1. Do not move shared-memory accesses across lock acquires/releases
   - Careful: A callee might do synchronization
   - Can relax this slightly [Effinger-Dean et al 2012]
2. Never add a memory operation not in the program [Boehm]
   - Seems like it would be strange to do this, but there are some non-strange examples (cf. homework problem)

Thread-local or immutable memory

- All these issues go away for memory the compiler knows is thread-local or immutable
  - Could be via static analysis, programmer annotations, or using a mostly functional language
- If the *default* were thread-local or immutable, maybe we could live with less/no optimization on thread-shared-and-mutable
  - But harder to write reusable libraries
- Most memory *is* thread-local, just too hard for the compiler to be sure